Weaving Ada 95 into the .Net Environment

Martin C. Carlisle, Ricky E. Sward, and Jeffrey W. Humphries
Department of Computer Science
United States Air Force Academy
{Martin.Carlisle,Ricky.Sward,Jeffrey.Humphries}@usafa.edu

ABSTRACT

This paper explains our efforts to add Ada to Microsoft's family of .NET languages. There are several advantages to weaving Ada into the Common Language Environment provided by the .NET environment. This paper explains our approach and current progress on the research. We provide the means to extract Ada specification files from Microsoft Intermediate Language (MSIL) code and compile Ada programs into MSIL.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications – *Ada, Object-oriented languages.*

D.3.4 [**Programming Languages**]: Processors – *compilers, interpreters, run-time environments*

General Terms

Algorithms, Design, Standardization, Languages.

Keywords

Microsoft .Net environment, Common Language Runtime, Ada 95, Just-in-Time compiling.

1. INTRODUCTION

Microsoft's .NET environment provides a large set of object-oriented libraries for application development, targeted especially for web-based applications. [11,12] It is an entirely new framework for programming Windows (and possibly other) machines. One of the key goals of .NET was to provide language interoperability. Ada shares similar goals, and was the first language to include mixed-language pragmas as part of its specification. Our A# project seeks to create a fully-interoperable environment for an Ada programmer to use .NET. Ada programmers will be able to use libraries written by other .NET programmers even if the libraries are written in other languages. Ada programmers will also be able to share their libraries with programmers using other languages.

This paper presents research being done at the Air Force Academy on the A# project. We will discuss our approach for compiling Ada into MSIL, extracting Ada specifications from MSIL, and our progress to date.

2. MICROSOFT'S COMMON LANGUAGE RUNTIME

In building the .NET Environment, Microsoft has found a way to provide language independent development coupled with platform independent execution. Their Common Language Environment (CLR) provides developers with a choice of several different programming languages such as C++, C#, Jscript, Visual Basic, and Perl. [10,11] The only requirement is that these languages must fit properly into the .NET environment. Each language must be compiled into the Microsoft Intermediate Language (MSIL) in order to run on separate platforms. The MSIL is then compiled using the Just-In-Time compiler specific to each runtime platform. [10]

The advantages of using the .NET environment include language independence and platform independence. The .NET environment also introduces garbage collection and support for versioning. Because of the language independence, any language that supports the CLR can support the same set of features. [10,11] The .NET environment also supports code-level access security where you can specify the level of security for code running on a .NET platform. [10] These advantages make the .NET platform an appealing target for Ada applications.

3. ADDING ADA TO THE .NET FAMILY

3.1 Overall Approach

To begin prototype work we used JGNAT and JBIMP to convert from Ada to MSIL (see Figure 1). The JGNAT tool developed by Ada Core Technologies compiles Ada into Java Byte Code. [4,8] The JBIMP tool developed by Microsoft as part of the J#

implementation compiles Java Byte Code into MSIL. [9] By using JGNAT to convert Ada to Java Byte Code and then using JBIMP to convert the Java Byte Code to MSIL, we developed a proof-of-concept prototype for the A# compiler.

In order for A# to tap into the rich resource of the .NET libraries, we needed to build Ada specification files for each library file. Our approach to this problem was to reverse engineer the MSIL code and recover the signatures of the functions and then build them into Ada specification files.

While doing the reverse engineering, we discovered that certain features of MSIL were not representable using the syntax provided by JGNAT. We therefore decided to rewrite JGNAT (into a tool we call "MGNAT") so that it outputs MSIL code directly, rather than going through the intermediate step of Java Byte Code.

To demonstrate the utility of the tools, we have ported RAPID, a GUI design tool to the .NET framework. [3]

3.2 Initial Prototype Work

In order to develop an initial proof-of-concept prototype, we used the JGNAT and JBIMP compilers to convert Ada to MSIL. [4,8,9]

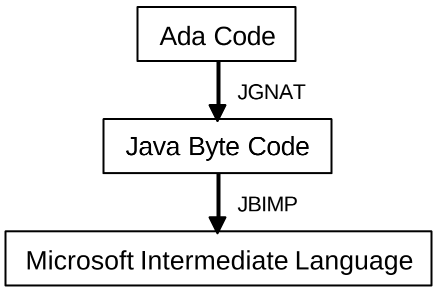

Figure 1 – Ada to Java Byte Code to MSIL

As shown in Figure 1, JGNAT converts the Ada code into a Java Byte Code .class file. This .class file is then converted to an MSIL file by JBIMP.

Using this process, we successfully converted an Ada "Hello World" program into MSIL. Initially, there was a problem with the conversion of Ada.Text_IO from Java Byte Code to MSIL. This error was a result of a bug in the implementation of JBIMP. A slight modification of the MSIL code solved the problem.

3.3 Recovering Ada Specifications from MSIL

One of the benefits of merging Ada into the .NET environment is the rich set of functions, web objects, and utilities available in the .NET environment libraries. In order to access these objects from Ada programs, we need a specification file for each MSIL library. Our approach in this case is to re-engineer each of the MSIL libraries and automatically generate an Ada specification for the library file. Ada programs can then make calls to the functions in the library specification file and the calls will be resolved when the MSIL code is compiled by the Just-In-Time compiler on the runtime platform.

The tool we have developed, msil2ada, takes in the MSIL as a text file and outputs Ada specification files. The MSIL text files can be generated from .NET dynamic link libraries (DLLs) using the ILDASM tool provided by Microsoft.

The first step in re-engineering MSIL code was to develop a grammar for MSIL. Given a grammar for MSIL, the AdaGOOP tool automatically generates a lexer, a parser, and the code for generating the MSIL decorated parse tree. [1]

While building the grammar for MSIL and using AdaGOOP to generate the parse tree, we modified AdaGOOP to also automatically generate tree traversal code. [2] Now, along with producing the lexer, parser, and parse tree for a given grammar, AdaGOOP also generates code to walk the parse tree. The developer specifies what action is to be done when walking the parses tree, but the changes required to specify this action are minimal. This generated code uses the Visitor pattern as a template. [6, 2] By using AdaGOOP, we were able to save large amounts of coding time; AdaGOOP automatically generated over 15,000 non-blank, non-comment lines of package bodies.

After building the grammar for MSIL and generating the parse tree, we used the generated tree traversal code to walk the tree and print out MSIL code. The effect was to parse in MSIL code, walk the parse tree and generate MSIL code. This validated the parser and ensured we had implemented the MSIL grammar properly. After this check, we used the tree traversal code to re-engineer the Ada specification file from the MSIL parse tree.

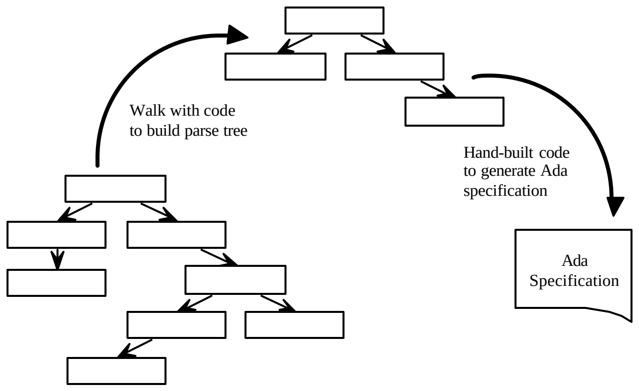
Walk with code to build parse tree
Hand-built code to generate Ada specification
Ada Specification

Figure 2 – MSIL parse tree to Ada specification

Figure 2 shows this process in greater detail. The parse tree shown on the left in Figure 2 represents a parse tree built from parsing an MSIL library file. The parse tree shown at the top of Figure 2 represents a subset of the MSIL parse tree, which includes only the signatures of namespaces, classes, methods, and fields. The implementation code from the library is not needed to generate the Ada specification file. Culling out this parse tree reduces the time required for operations on the parse tree and significantly reduced compilation time while developing the actions for the culled tree. (On our development machine, compiling the action file for the full tree required 110 seconds, compared to 4 for the action file for the culled tree). The tree traversal code generated by AdaGOOP is used to walk the tree and generate the minimized parse tree. Once this tree has been built, hand-built code is used to walk the minimized parse tree and generate the Ada specification file (shown at the bottom right in Figure 2).

We have tested this re-engineering of MSIL code on a C# DLL file named TimeLibrary.dll. This file is an example from Deitel & Deitel, C# How to Program. [5] We have successfully converted TimeLibrary.dll into an Ada specification file. Figure 3 shows an excerpt from the TimeLibrary.cs C# file and Figure 4 shows the corresponding Ada specification file.

```
// TimeLibrary.cs
// Placing class Time3 in an assembly for
reuse.
using System;
namespace TimeLibrary
{
   // Time3 class definition
   public class Time3 : Object
   {
      private int hour;    // 0-23
      private int minute;  // 0-59
      private int second;  // 0-59
   // Time3 constructor: hour and minute
supplied, second
      public Time3( int hour, int minute )
      {
         SetTime( hour, minute, 0 );
      }
      // property Hour
      public int Hour
      {
         get
         {
            return hour;
         }
         set
         {
            hour = ( ( value >= 0 && value <
24 ) ? value : 0 );
         }
      } // end property Hour
…
```

Figure 3 – TimeLibrary.cs

```
pragma Extensions_Allowed(On);
with MSSyst.Object;
with MSIL_Types;
use MSIL_Types;
with type TimeLibrary.Time3.Ref is access;
with type MSSyst.String.Ref is access;
package TimeLibrary.Time3 is
   type Typ;
   type Ref is access all Typ'Class;
   type Arr1 is array(Natural range <>) of
Ref;
   type Ref_Array is access all Arr1;
   type Typ is new MSSyst.Object.Typ with
record
      null;
   end record;
…
   function new_Time3(
      This : Ref := null;
      hour : Integer;
      minute : Integer) return Ref;
   function get_Hour(This:access Typ)return
Integer;
   procedure set_Hour(
      This : access Typ;
      value : Integer);
…
```

Figure 4 – Ada specification for TimeLibrary

Using the TimeLibrary Ada specification file, we built an Ada main program with calls to functions in the library file.

We then used msil2ada to translate the following standard .NET DLLs: system.dll, system. windows.forms.dll, system.drawing.dll, mscorlib.dll. These DLLs contain all of the necessary classes for doing basic user-interface design (including dialogs and windows with menus, buttons, text boxes and other standard components).

For simplicity, we adopted many of the same conventions used by jvm2ada (the tool that translates Java class files into Ada specifications). In particular, we used the same techniques for handling interfaces, circular type dependency, and constructors. However, we developed our own techniques for handling ValueType and Enumeration, which were not present in Java.

3.3.1 Interfaces

The designers of JGNAT chose to handle Java *interfaces* by specifying a type which implements interfaces as parameters of the type, as shown in Figure 5.

```
  type Typ(
        I_IContainerControl :
            IContainerControl.Ref;
        I_ISynchronizeInvoke :
            ISynchronizeInvoke.Ref)
   is new ContainerControl.Typ(
        I_IContainerControl =>
            I_IContainerControl) with record
      null;
   end record;
```

Figure 5 – Ada specification for class with interfaces

The class in Figure 5 implements two interfaces, IContainerControl and ISynchronizeInvoke. Since the parent class also implements IContainerControl, this appears again after the name of the parent type.

When building msil2ada, we had to deal with .Net *interfaces*, which are similar to Java interfaces. We chose to handle them in the same fashion as JGNAT and build types which implement the interfaces as parameters of the type. To do this, msil2ada automatically discovers all of the interfaces in the parent classes (if the parent class is present in the input) and generates the appropriate references. Unfortunately, there are instances in the MS DLLs where a class has its parent in another DLL and msil2ada fails to generate the necessary interface references, yielding a compilation error on the specification. We have instrumented msil2ada to recognize the most troublesome examples of this (the Control class in System.Windows.Forms.dll) and output the correct code for this case.

3.3.2 Circular Type Dependency

In Ada 95, types that are mutually dependent must be declared in the same package. Mutually dependent types abound in the .NET and JVM libraries, so this approach is impractical. As was done in JGNAT, we implement the “with type” context clause in A# to handle such mutual dependencies. Figure 4 shows an example of the “with type” clause.

3.3.3 Constructors

As in JGNAT, constructors are marked with a special pragma, MSIL_Constructor. They take a single parameter, which defaults to null. Normally, to create an object of the class, you just call the constructor with no parameters. If you want to create your own constructor for a type that is a descendant of a .NET type, then the syntax becomes somewhat counterintuitive. Figure 6 shows that the first step is to call the parent constructor with “This” as a parameter (even though it appears that “This” has never been given a value, the allocation is added implicitly by the compiler). Then, “This” is returned. Between the “begin” and “end” additional initialization required for the child class may be performed.

```
function New_MenuItem(This : Ref := null)
         return Ref is
      Super : MenuItem.Ref :=
         MenuItem.New_MenuItem(
            MenuItem.Ref(This));
   begin
      return This;
   end New_MenuItem;
```

Figure 6 – Constructor for child of .NET class

3.3.4 ValueType

The Java Virtual Machine, includes only base types (such as integer and float) and class references. The MS .NET platform allows for the creation of types that are passed by value (hence the name ValueType) instead of by reference. To resolve this, we have added the reserved word "ValueType". We use this type name instead of "Typ" and "Ref" as shown in Figure 4. For example, a class reference to the Time3 class is named TimeLibrary.Time3.Ref, but a Point (from System.Drawing.dll) is named MSSyst.Drawing.Point.ValueType. If a type is so named, then the compiler will generate code for it in accordance with the MSIL calling conventions for ValueTypes. Additionally, there is no pointer for this type, so the modified context clause to handle mutual dependencies ends with "is tagged" instead of "is access".

3.3.5 Enumeration

The .NET Framework provides *enumeration types*, which are child classes of Enum (which is a child of ValueType). We initially mapped these to Ada enumeration types. However, unlike Ada enumeration types, .NET enumerations can have multiple names corresponding to the same value, and, in certain cases, can be combined to create values that have no name. For example, in the FontStyle enumeration, **Bold** and *Italic* are listed separately, but they can also be added together to create a ***Bold Italic*** style, although this is not listed in the enumeration. On closer observation, these appear to correspond more directly to named constants. For now, we provide a function "+" for performing such combinations. Currently, all of the attributes of Ada enumerations ('Pos, 'Succ, etc.) do not work correctly on these types. In a future version, we may implement these as named constants.

3.4 Compiling Ada Directly to MSIL

In order to compile Ada directly into the Microsoft Intermediate Language (MSIL), we have re-written the JGNAT compiler into a tool we call MGNAT. MGNAT compiles a modified Ada syntax and outputs MSIL directly. We refer to the modified Ada syntax as the A# language. Many of the compiler design issues were already addressed in our discussion of msil2ada, but there were a couple of issues that were unique to the compiler.

One design goal was to make programming .NET from Ada as transparent as possible. We made two language changes in A# in order to simplify using the compiler: allowing object.method syntax for method calls, and making conversions from Ada strings to .NET strings implicit.

3.4.1 Object.Method syntax

Ada 95 has often been criticized for making the syntax of dispatching method calls the same as the imperative procedure calls. Consider the following example from C#:

```
Window1.ResetSecurityTip(
   true);
```

The same call in Ada 95 would appear as:

```
MSSyst.Windows.Forms.ResetSecurityTip(
   This      => Window1,
   modalOnly => True);
```

This can become quite tedious, as packages tend to be nested deeply. One alternative is to add use clauses for all of the packages. Still, if the user is trying to follow the examples in the help files, they need to constantly convert to the Ada syntax. We have modified the compiler to allow the same object.method syntax:

```
Window1.ResetSecurityTip(
   modalOnly => True);
```

We have also performed this modification to the Windows 3.15 version of GNAT. It required only 127 non-blank, non-comment lines of Ada code. We expect this syntax will make it easier for students to understand object-oriented programming. Since we also support the standard Ada 95 syntax, we still have the nicer Ada syntax for operators (x+y instead of x."+"(y)).

3.4.2 Implicit string conversions

Both the JVM and .NET provide string classes. In JGNAT, Ada strings were incompatible with these classes and a conversion operator, "+", was provided. This leads to code as the following:

```
Console.Writeline(+"Hello world");
```

While this is consistent with Ada's strong typing, we felt that it made the coding seem awkward. Hence, if the user has a use clause on MSSyst.String, the compiler will automatically insert the conversion operator. A conversion operator is also provided to go from .NET strings to Ada strings; however, it is not implicitly added. An example of its use is below.

```
declare
   Y : String := +Console.ReadLine;
begin
```

3.5 Porting RAPID to .NET

In order to test our implementation of MGNAT, we ported RAPID [3] to the new A# language. Although in general we found the .NET port to be easier and require less code than the JVM port, there were three features which .NET would not support without considerable effort. RAPID has been ported to Tcl/Tk, Gtk, JVM and .NET. .NET was the only platform that didn't fully support the RAPID library.

First, .NET doesn't contain a routine for making a bell sound. A newsgroup respondent to a question on this suggested using the DirectSound interface (which would have required a separate download and install).

Second, .NET radio buttons can not be distinguished into groups without placing each group in a GroupBox. (On other platforms, the user can specify what radio buttons should go together without adding another GUI element).

The most frustrating limitation was that menus could only be attached to a form. (In the other versions of RAPID, a WYSIWYG menu is displayed for the form being designed).

5. FUTURE WORK

At this point, there are many opportunities for improvement of the MGNAT compiler and msil2ada. First, there are several .NET types that are not currently representable. Msil2ada does not currently address marshalling objects for the COM interface, and the resulting code fails to compile for functions that return unconstrained arrays.

Second, .NET enumerations need to be readdressed. Although currently we map these to Ada enumerations, the Ada enumeration attributes aren't working. A possible solution is to switch these to named constants, which will require modification of both msil2ada and the compiler.

Finally, the Ada libraries have not yet been compiled with this compiler. We currently have translated jgnat.jar (the Java compressed class files) into jgnat.dll using the Microsoft conversion tool JBIMP. This introduces an unnecessary dependence on the Visual J# libraries. Compiling with the new compiler will require translating all of the JVM library calls to .NET calls.

5. CONCLUSION

The advantages of the .NET environment with its language interoperability and platform independence make it an attractive target for Ada developers. We have built a compiler for an Ada language for .NET, which we call A# that translates directly into the Microsoft Intermediate Language. To demonstrate the utility of the tool, we have used it to compile RAPID, a GUI design tool. While there remains significant room for improvement, this tool will be useful for a large class of Ada applications, and will allow Ada developers access to the .NET Framework.

6. REFERENCES

[1] Carlisle, Martin C., *An Automatic Object-Oriented Parser Generator for Ada*, Ada Letters, Volume XX, Number 2, June 2000, pp. 57-63.

[2] Carlisle, Martin and Ricky Sward, *An Automatic "Visitor" Generator for Ada*, Ada Letters, Volume XXII, Number 3, September 2002, pp. 42-47.

[3] Carlisle, Martin and Pat Maes, *RAPID: A Free Portable GUI Designer for Ada*, Proceedings of SIGAda '98, ACM, 1998.

[4] Comar, Cyrille, Gary Dismukes, and Franco Gasperoni, *Targeting GNAT to the Java Virtual Machine*, Proceedings of the Tri-Ada 97 Conference, St Louis MO, Nov 9 – 13, 1997.

[5] Deitel, H. M., P. J. Deitel, J. Listfield, T. R. Nieto, C. Yaeger, and M. Zlatkina, *C# How to Program*, 2002, Prentice Hall, Upper Saddle River, NJ 07458

[6] Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1995, Addison-Wesley, Reading, MA, pgs 331-344.

[7] Gasperoni, F. and G. Dismukes, *Multilanguage Programming on the JVM: The Ada 95 Benefits*, Ada Letters, Volume XX, Number 4, December 2000.

[8] Gosling, J., B. Joy, and G. Steele. *The JavaÔ Language Specification*, Addison-Wesley, 1996.

[9] Jepson, Brian, A Visual J# .NET Primer, http://www.oreillynet.com/pub/a/dotnet/2001/10/15/jsharp.html

[10] Platt, David S. *Introducing Microsoft .NET*, 2001, Microsoft Press, Redmond, WA 98052-6399.

[11] The .NET Runtime Environment. See http://www.microsoft.com/net/.

[12] Weiss, Aaron, *Microsoft's .NET: Platform in the Clouds*, ACM White Paper, Dec 2001.